Scheduling Obfuscation: Analyzing the Endeavor of Deterring Timing Inference Attacks on Real-Time Systems

Simon Langer
simon.langer@fau.de
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

ABSTRACT

With the current trend to more autonomous decision-making (e.g. automation in industry, mobility, infrastructure), the need for highly complex security-critical real-time systems increases. To achieve this complexity in an affordable manner, the use of more external (hard-/software) components – with potential flaws in security and lack of full trustworthiness – is a requirement. Information on when a victim task was or will be executed can significantly aid a would-be attacker (already with a foot in the door) in succeeding without being easily detected (timing inference attacks via scheduler side-channels).

This paper summarizes several recently developed attack algorithms (ScheduLeak, DyPS) and randomization-based deterrence strategies (TaskShuffler, TaskShuffler++), then discusses their limitations – both in terms of practicality as in suitability for decreasing vulnerability – and finally touches on alternative concepts that seem more promising.

1 TIMING INFERENCE ATTACKS AND WHY YOU SHOULD CARE ABOUT THEM

One of the key components of a real-time system (RTS) is its scheduling infrastructure: RTS are built to meet strict deadline guarantees – this necessitates reliability and predictability in their scheduling algorithms. As minimum requirement in terms of predictability, designers are obliged to ensure that the intended task set is schedulable by the selected scheduling algorithm. Both fixed-priority (FP) using rate monotonic priority assignments and earliest-deadline first (EDF) scheduling provide very high reliability and predictability in a preemptive RTS (fundamental paper: [8]).

Timing inference attacks exploit the aforementioned properties, i.e. timing information can provide an (unintended) source of knowledge (= side-channel) to an unprivileged attacker task. This allows for targeted attacks on data transfer, control signals, or general system resources (e.g. cache side-channels).

1.1 ScheduLeak on FP

Due to their rather strong isolation from the outside world, RTS were long considered to be immune to typical cybersecurity issues [7, 10, 14]. However, more complex and interconnected systems require a deeper analysis of potential attack vectors [19].

1.1.1 *System and Adversary Model.*

The ScheduLeak algorithm presented by Chen et al. in [4] assumes a mixed periodic, sporadic and non-real-time task set, executed on a uniprocessor fixed-priority preemptive RTS. Task periods are considered distinct and deadlines equal to the respective periods. With negligible task release jitter[1], the time between two consecutive task executions of a periodic task is constant.

ScheduLeak's goal is to leak timing information of one periodic victim task (=: τ_v) from the viewpoint of a lower priority, unprivileged observing attacker (=: τ_o) with access to a system timer. The observer is hereby assumed to be a periodic task, as that is a more restrictive premise: A sporadic observer could additionally delay its next release longer than its set minimum inter-arrival time thus allowing for greater variety in relative execution times; it can simulate periodic release behavior by setting the next release to the same value on every execution. Typical monitor-and-adjust duties (like controlling output with pulse-width modulation) that do mission-critical work have a periodic nature linked to physical properties; the attacker knows these in advance, hence the victim's period (=: $period(\tau_v)$) as well.

1.1.2 *Schedule: Leak!*

Measuring Execution Intervals of τ_o. A part (=: λ) of τ_o's worst-case execution time (=: $WCET(\tau_o)$, upper bound on task's execution time) is dedicated to repeatedly polling the current system time and recording abnormally high differences between two consecutive calls; consequently, preemptions of τ_o by higher priority tasks (=: $hp(\tau_o)$) are detected.

Some time ($WCET(\tau_o) - \lambda$) needs to be reserved to maintain τ_o's original functionality and leave headroom for other calculations.

Organizing Execution Intervals of τ_o. To better analyze the execution intervals, Chen et al. [4] split the time axis into segments of equal length $period(\tau_v)$, visualized as horizontal rows in a *schedule ladder diagram*. A recurring time slot (width: 1 time unit) is represented as a vertical column in the diagram; these slots are the main tools for extracting timing information.

Thanks to τ_v's periodicity, we know τ_v will be released at $n \cdot period(\tau_v) + arrivalT(\tau_v), n \in \mathbb{N}$. Splitting the time axis into segments of equal length $period(\tau_v)$ is equivalent to taking the remainder of the release time divided by $period(\tau_v)$: $(n \cdot period(\tau_v) + arrivalT(\tau_v))\ mod\ period(\tau_v) = arrivalT(\tau_v) \rightarrow$ is constant. So τ_v will always arrive at the same time slot for each row =: $slot(\tau_v)$.

Since $priority(\tau_v) > priority(\tau_o)$, τ_o cannot run while τ_v's execution for this $period(\tau_v)$ is still pending: $slot(\tau_v) \notin slots(\tau_o)$.

This is also the case for slots immediately following $slot(\tau_v)$ until τ_v's best-case execution time (=: $BCET(\tau)$, lower bound on task's execution time) is reached: τ_v will at least run for its BCET before any task of lower priority (including the observer) can execute, i.e.

[1] task release jitter = maximum possible deviation between planned theoretical release time and actual physical release time of a periodic task

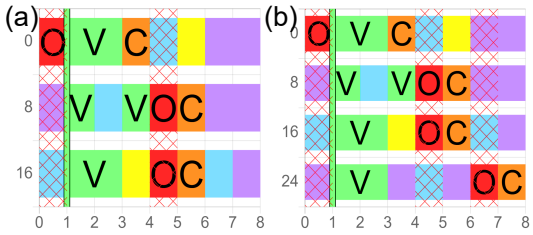

Figure 1: Screenshot of the interactive simulation *ScheduLeak in Action* (https://www.simonlanger.com/uni/KvBK/?render=2): This schedule ladder diagram divides time (arranged in row-major order) into rows of length 8 since the period of the victim task (in green, "V") is 8. As a result, the victim's arrival time can be visualized as the green vertical line in column 1. The attacker task is assigned two colors and letters to separate the time it uses for observations (in red, "O") and the time it uses for other calculations and keeping up its service (in orange, "C") – in this example, the ratio is equal, i.e. $\lambda = WCET(\tau_o) - \lambda$. The other colors represent additional tasks in the system. In (a), the time slots $[0, 1)$ and $[4, 5)$ have already been permanently eliminated (crossed-out columns) since the observer was allowed to run at least once during each of them. In (b), the observer has run once again; now, the time slot $[6, 7)$ can also be eliminated. At this point, the longest contiguous time slot is $[1, 4)$; its beginning is 1, which is the arrival time of the victim relative to the observer (i.e. the attacker's goal).

τ_v will only be preempted by tasks of higher priority and afterwards still has to finish its own execution before it is the observer's turn.

In short: **Any time slot during which τ_o was allowed to run at least once can be permanently ruled out**, we gradually narrow down the range of possible $slot(\tau_v)$-candidates; this is visualized in Figure 1.

Deducing future arrivals of τ_v. The next (obvious) step is to choose the most promising $slot(\tau_v)$-candidate – this is necessary because system parameters influence when τ_o could run and observe (e.g. limitations by: many higher priority tasks other than τ_v, $period(p_o)$ shares many factors with $period(p_v)$, short $WCET(p_o)$, small λ – details: *coverage ratio* in [4]).

As result-slot, we choose the longest consecutive $slot(\tau_v)$-candidate interval and call its start-slot-number $slotNum(\tau_v)'$ (at least until $BCET(\tau_v)$ is over, τ_o cannot run; empirically proven to be a good choice [4]).

Using the (hopefully correct) start-slot-number $slotNum(\tau_v)'$, we predict future arrivals of τ_v:

$$arrivalT(\tau_v)' = (t_{attackStart} + slotNum(\tau_v)') \bmod period(\tau_v)$$
$$futureArrivalTimes(\tau_v)' = arrivalT(\tau_v)' + period(\tau_v) \cdot n, n \in \mathbb{N}$$

1.2 DyPS on EDF

ScheduLeak's approach of gradually narrowing down possible slot-candidates can be transferred from a fixed-priority to a dynamic-priority (EDF) environment – demonstrated by Chen et al. [3] in their DyPS scheduler side-channel attack.

1.2.1 System and Adversary Model. As mentioned above, DyPS – in contrast to ScheduLeak – focusses on an EDF RTS with a mixed task set of sporadic, periodic and aperiodic tasks; conflicts of multiple tasks having the same (earliest) deadline are resolved by randomly choosing one of them to execute first.

1.2.2 Dynamic-Priority: ScheduLeak! ScheduLeak's Step 1 "Measuring Execution Intervals of τ_o" is only valid if $priority(\tau_v) > priority(\tau_o)$, which is not always the case with dynamic EDF.

Measuring $taskPhase(\tau_o)'$. $taskPhase(\tau)$ is the projection of $arrivalT(\tau)$ onto $[0, period(\tau))$ using $mod\ period(\tau)$.

To calculate the $taskPhase(\tau_o)'$, we need a timestamp of when τ_o was released, but we can only query the system time while τ_o is running. Immediately on startup of each job, the current time (=: $startTime$) is saved and compared to the previous best approximation $bestStartTime$; for this purpose, $startTime$ is arithmetically moved back to $bestStartTime$'s period by subtracting the n periods of τ_o's executions that happened in the meantime. If τ_o was at least once allowed to run immediately upon release, the approximated task phase will be exactly correct.

Computing when τ_v is observable by τ_o and measuring execution intervals of τ_o. ScheduLeak's Step 1 "Measuring Execution Intervals of τ_o" is restricted to the Interval $[lastArrivalT(\tau_o)', lastArrivalT(\tau_o)' + period(\tau_o) - period(\tau_v))$

If $lastArrivalT(\tau_o)' := startTime - ((startTime - taskPhase(\tau_o)') \bmod period(\tau_o))$ is correct, **the observer only measures slots during which the victim did not want to run** – so those can be ruled out: $slot(\tau_v) \notin measuredSlots(\tau_o)$.

Deducing future arrivals of τ_v. As in ScheduLeak, the longest consecutive $slot(\tau_v)$-candidate interval is chosen. Both task phase and future arrival times can then be calculated correspondingly.

2 SCHEDULE-BASED MOVING TARGET DEFENSE DETERRENCE APPROACHES

After having delineated the attacker side of scheduler side-channels, in the following, this paper analyzes and discusses state-of-the-art scheduling obfuscation techniques.

2.1 TaskShuffler

The key idea of TaskShuffler (introduced by Yoon et al. [21]) is to obfuscate a FP-schedule by allowing for bounded priority inversions – i.e. under certain circumstances, a task is allowed to run even though it is not the highest priority one in the ready queue – and picking one of the possible tasks at random on each scheduling decision.

2.1.1 System Model. TaskShuffler schedules a task set of only periodic tasks with distinct priorities (assigned via the Rate Monotonic Algorithm [8]; an idle task with infinite period and WCET is included), schedulable by FP preemptive scheduling on a single-core system. Accordingly, the worst-case response time (derived using iterative response time analysis, starting with $prevWCRT(\tau) := WCET(\tau)$, until $prevWCRT(\tau) = WCRT(\tau)$)

$$WCRT(\tau) = WCET(\tau) + \sum_{\tau_j \in hp(\tau)} \left\lceil \frac{prevWCRT(\tau)}{period(\tau_j)} \right\rceil \cdot WCET(\tau_j)$$

is $\leq deadline(\tau)$ for every task in the task set.

2.1.2 Tasks: Shuffle! In order not to miss deadlines, it is necessary to limit priority inversions – TaskShuffler does this by calculating a worst-case maximum inversion budget (=: $maxInv(\tau)$) offline for each task and keeping track of the remaining inversion budgets of every task (=: $remInv(\tau)$) at runtime.

Calculating priority inversion budgets (offline). When allowing for a priority inversion by executing a task of lower priority

than τ ($\in lp(\tau)$) for some time while τ is ready, τ might take more than $WCRT(\tau)$ to complete: In addition to being interrupted by higher priority tasks that are released while τ is being executed (they preempt τ), there can also be higher priority tasks that were deferred by the priority inversion – they need to execute while it would be τ's turn to not miss their deadlines (*back-to-back hit*).

The total worst-case interference is calculated by adding up the number of times each higher priority task can be released within the timespan of $deadline(\tau)$ plus 1 for a deferred instance each and multiplying it by their respective WCET:

$$interference(\tau) = \sum_{\tau_j \in hp(\tau)} \left(\left\lceil \frac{deadline(\tau)}{period(\tau_j)} \right\rceil + 1 \right) \cdot WCET(\tau_j)$$

So, the **worst-case maximum inversion budget (maxInv)** is calculated by subtracting both the interference and WCET of each task from its deadline:

$$maxInv(\tau) = deadline(\tau) - (WCET(\tau) + interference(\tau))$$

$remInv(\tau)$ is reset to $maxInv(\tau)$ when τ is released, and decremented by 1 for every time step that any of $lp(\tau)$ executes while τ is ready. This ensures all tasks hit their deadlines, as long as $\forall \tau \in taskset(maxInv(\tau) \geq 0)$.

Dealing with negative $maxInv(\tau)$. Additional measures are required so that a task set with one or more tasks with $maxInv(\tau) < 0$ remains schedulable: No lower priority tasks can run while τ is ready – this is not sufficient; a future job τ might still be significantly delayed by deferred executions of $hp(\tau)$ that were ready before this future job arrived, ultimately causing τ to miss its deadline. For each τ with $maxInv(\tau) < 0$: **As long as any higher priority task is ready, the execution of all lower priority tasks is forbidden (exclusion policy)**.

Random scheduling. (An example is visualized in Figure 2.) The randomized scheduling decision is made by first collecting all ready tasks that can run without violating any of the above mentioned constraints and second picking one from the list (each task has an equal probability to be selected). The next decision is made when any remaining inversion budget of a higher priority task reaches 0, a new job arrives, or the selected task finishes – this can also happen earlier if fine-grained switching is used.

2.2 TaskShuffler++

TaskShuffler's approach of permitting bounded priority inversions can be extended to support sporadic tasks and offer better randomization by doing more calculations online – demonstrated by Yoon et al. [20] in their TaskShuffler++ algorithm.

This paper will focus on the approximate version of TaskShuffler++ since the exact one requires online iterative computations and thus is not suitable for real-world use (the authors of [20] point this out themselves).

2.2.1 System Model. In contrast to TaskShuffler, sporadic tasks are permitted.

2.2.2 Testing inactive tasks.

Primary Test. If the sum of already released tasks' remaining WCETs (=: $remWCET(task)$) plus the sum of WCETs of tasks that will be released from now (=: t) until τ can be released again

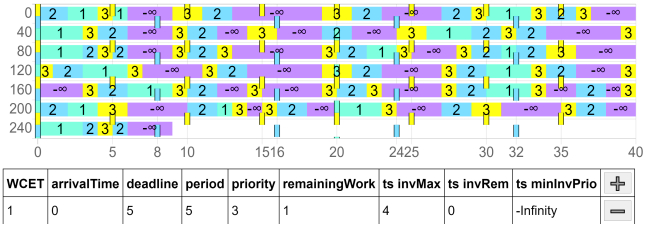

Figure 2: Screenshot of the interactive simulation *TaskShuffler in Action* (https://www.simonlanger.com/uni/KvBK/?render=3): Each row in this diagram represents one hyperperiod (i.e. the smallest common multiple of all tasks' periods, here: 40); time is arranged in row-major order. With FP-scheduling, every row of this diagram would be exactly the same; TaskShuffler's randomization, though, manages to significantly reduce the predictability of task executions. Tasks are labeled with their respective priorities – from highest to lowest: $3 > 2 > 1 > -\infty$ (idle task). Note that two consecutive executions of the same task are displayed as one contiguous horizontal segment. At the point in time this snapshot was taken, the remaining inversion budget of the highest priority task is 0 (ts invRem) while it has remaining work, so it will definitely execute in the next time slot; this ensures that the task does not miss its deadline (depicted by the vertical dashed yellow line in column 10) – once the job is completed and task 3's next job released, the remaining inversion budget will be reset back to task 3's maximum inversion budget ts invMax = 4.

(minimum inter-arrival time =: $interArr(\tau)$) is less than the next point in time when τ can be released again (=: $nextArr(\tau)$), there exists a budget (the difference) for priority inversions.

Secondary Test. If the primary test fails, we do not know for sure if priority inversions are possible. So, the secondary test gives it another shot (but also without any guarantees). Its idea is to a) move all tasks that release before $nextArr(\tau)$ to release together at the latest possible time and b) move remaining task executions to that same point in time (:= $tBad$). The WCETs of a) and remWCETs of b) are added up and compared to $(nextArr(\tau) - tBad)$[2].

A positive difference indicates a need for slack in τ as not everything can be done before $nextArr(\tau)$. This slack (:= $slack(\tau)$) for every task can be calculated offline by gradually increasing it up from 0, so that the sum of WCET, slack and interference from higher priority tasks (*not* including deferred executions, that is the mission of the above moving-and-adding) $\leq deadline(\tau)$.

2.2.3 Testing active tasks. The maximum interference (=: $interference(\tau)$) by higher priority tasks until τ reaches its deadline is computed by adding up the following for each task $\in hp(\tau)$: remaining WCET, full WCETs possible until deadline and part of an execution that started before the deadline but will not finish before the deadline is reached.

Analogous to TaskShuffler, $remInv(\tau)$ is initialized to $currMaxInv(\tau) = deadline(\tau) - (WCET(\tau) + interference(\tau))$.

Random scheduling. As in TaskShuffler, candidacy tests (this time the more complex ones outlined above) are performed on each task in the ready queue (an individual test for active/inactive tasks is run for each $\in hp(\tau)$), yielding a collection of candidate tasks.

The randomized selection is – in contrast to TaskShuffler – weighted by the ratio of $remWCET(\tau)$ to the time until τ's deadline: In the long run, this distributes task executions more uniformly.

[2]Figure 5 in [20] depicts this process of moving and adding.

2.3 Related Work: REORDER, Slot-Level Randomization

2.3.1 Idea of REORDER. The REORDER (REal-time ObfuscateR for Dynamic SchedulER) protocol [2] transfers the Task-Shuffling approach [21] from FP to EDF. The main difference lies in how to calculate maximum inversion budgets: $maxInv(\tau) = deadline(\tau) - WCRT(\tau)$.

To compute the global WCRT, local WCRTs at specific release times (of τ) are calculated up to t_{max}[3]. Finding the local WCRT primarily involves determining the workload: τ's own work plus interference (i.e. how much higher priority work has piled up at time t).

To find the interference, for each task τ_j whose priority is greater or equal to τ's priority (i.e. $deadline(\tau_j) \leq t + deadline(\tau)$), the maximum number of τ_j executions that can interfere with τ is calculated, multiplied by $WCET(\tau_j)$ and added up.

When does an execution of τ_j interfere with τ? It arrives before τ's deadline is reached and its priority is greater than τ's (i.e. τ_j's deadline is earlier than τ's). So, the number of τ_j's that interfere is the minimum of these two values: maximum number of times τ_j can arrive within one $deadline(\tau)$ and number of τ_j arrivals that can happen while the priority of each of them remains greater to τ's [16]; in both cases, back-to-back hit must be considered.

2.3.2 Idea of Slot-Level Randomization. Fundamentally different from all of the attack-/defense scenarios covered above, Slot Shifting algorithms schedule on time-triggered systems – i.e. the scheduler is called at fixed intervals (=: *slots*): Offline computed scheduling tables include available leeway within the schedule (spare capacities) that can be used e.g. by aperiodic tasks [5].

Instead of using the leeway to include aperiodic tasks, it can be used to randomize when periodic ones execute: Jobs that share the same deadline are grouped as a capacity interval; for each capacity interval, the remaining leeway is tracked – if it is positive, any job may execute in the next slot without violating any deadlines [7]. Krüger et al. did not run performance evaluations on a simulator or real hardware, but merely reference a case study of regular, non-randomizing Slot Shifting and argue that adding randomization does not significantly increase scheduler overhead.

3 DISCUSSION OF PRACTICALITY AND WHY YOU SHOULD CONSIDER ALTERNATIVE STRATEGIES

Unfortunately, all of the presented moving target defense approaches (Section 2.1, Section 2.2, Section 2.3) imply drawbacks in terms of applicability and usefulness – these will be discussed hereafter, followed by some alternatives.

3.1 Restrictive Prerequisites for Shuffling

Directly implementing TaskShuffler(++), Slot-Level Randomization or REORDER requires meeting several (rather restrictive) system model assumptions stated in their respective papers.

All of them **only work on single-core platforms** while, as mentioned in the abstract, especially RTS with higher performance and complexity are more at risk as they include more external components and run more code. The gradual transition to multi-core can be best exemplified by state-of-the-art heterogenous autonomous driving hardware by Tesla [17], and NVIDIA [12]. Accordingly, tackling the limitation to uniprocessor systems is receiving more attention recently, e.g. the original TaskShuffler was adapted to homogenous multi-core platforms [1][4].

Obfuscation techniques severely limit a system designer's choices on how to efficiently, cost-effectively build an RTS for the job – all except TaskShuffler++ **require the exclusive use of periodic tasks**. Handling an I/O-interrupt is a typical sporadic task; to realize this functionality in a periodic manner entails (in most periods) wasting the allocated, rather large time budget. Consequently, planning and implementation costs increase or even a faster (and more expensive, power-hungry) CPU might be needed.

Moreover, randomizing task executions generally **creates additional overhead** in the system: Higher scheduler complexity implies longer runtime and generating shuffled execution patterns involves calling the scheduler more often, increasing context switches. That's why [20, 21], and [2] conducted thorough evaluations not only of the respective shuffling performance, but also of the induced performance hit. They typically show a measurable, yet manageable impact; regular TaskShuffler, for instance, increases context switches by a factor of 1.1 to 1.8 on average (heavily dependent on CPU utilization and if fine-grained switching is enabled). The lack of performance evaluation in [7] raises doubts of the real-world viability of the suggested Slot-Level Randomization; Nasri et al.'s independent implementation substantiates these doubts as they observed an increase of preemptions by up to three orders of magnitude [11].

Common to all described techniques, the significant **increase in scheduler complexity reduces (emergency) flexibility**: Hence, e.g. inserting a new, unknown (to the scheduler) real-time job at runtime is no longer a (theoretical) option; assuming adequate headroom, you could livepatch a control system to adapt to unforeseen circumstances by running a new binary with high priority (FP) or with short deadlines (EDF) – no restart of the system or scheduler (creating downtime) is required. Conceptually, FP and EDF schedulers get the most important (or most time-critical) work done first, which is foiled by (mis)using this headroom in order to mix timings up.

3.2 Sacrificing Stability for Security?

Not executing the most important/time-critical tasks first (see previous paragraph; e.g. in Figure 2 the most important task (3) only barely hits its deadline in row 80, columns 5, 10, 20, 25, 35) can create unnecessary, avoidable stability and reliability risks.

The inferred WCETs or deadlines might not hold under specific circumstances: **Issues caused by an underestimation of WCET or overestimation of the real-world deadline are amplified** when using schedule randomization of any kind, as these algorithms use the calculated headroom (e.g. priority inversion budgets in TaskShuffler) for randomizing task executions and leave none (or little in case of task release jitter) for unexpected situations. For

[3] $WCRT(\tau) = max\{WCRT\_at\_Time(t, \tau)\}$; for t_{max} see [2]

[4] However, as a positive side effect, multi-core can contribute to more secure RTS (Section 3.4).

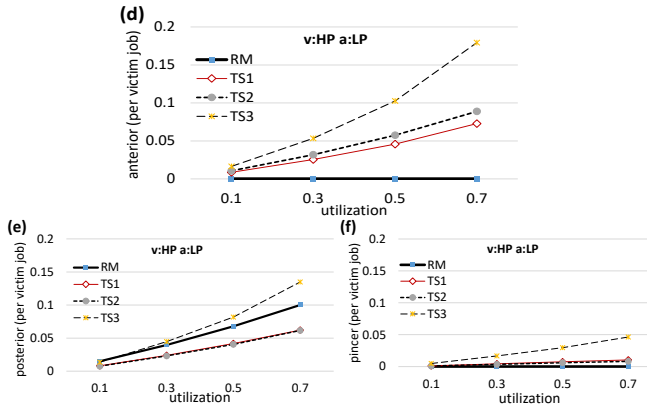

Figure 3: Exemplary situation of 10 tasks with the victim having the highest priority, the attacker the lowest; the x-axis denotes CPU utilization, the y-axis denotes Nasri et al.'s Attack Success Ratio (ASR), i.e. the ratio of successful attack constellations compared to the overall number of considered victim task's jobs. A simple FP-scheduling approach (labeled as RM for Rate-Monotonic priority assignments) is immune to anterior attacks (d) (and accordingly also to pincer (f) requiring both posterior and anterior); when introducing TaskShuffler (TS1 = TaskShuffler without idle time scheduling, TS2 = TaskShuffler with idle time scheduling, TS3 = TaskShuffler with idle time scheduling and fine-grained switching), this immunity is destroyed. In the case of posterior attacks (e), enabling shuffling has minor effects in either direction (all three graphs from [11]).

instance, an attacker could provoke cache misses of a critical task, slowing it down past its WCET, and due to shuffling, it misses its deadline; in contrast, without shuffling (in a FP-environment), it would have finished well before the deadline approaches, having a high priority.

Another obstacle in a designer's path to a stable RTS with schedule randomization is the (current) **incompatibility** of randomization **with features** that modern real-time operating systems (RTOS) offer **like** support for **synchronization** [2, 20]. This forces every system designer to somehow work around this restriction, which can easily lead to some systems ending up in a deadlock or logical errors due to imperfect pseudo-synchronization.

3.3 Potentially Counterproductive Outcome

The underlying philosophy of moving target defense – i.e. increase complexity and costs for the attacker, in other words: **deterrence** – is **fundamentally different from** employing preventive mechanisms – i.e. blocking attackers upfront, in other words: **protection**. Implementing randomization techniques can create a deceptive sense of security that can lead to omitting other security measures (see Section 3.4).

Quantifying randomness is a crucial (and problematic) part in the evaluation of TaskShuffler(++) [20, 21], and REORDER [2]. Each of the three papers defines its own metric: In TaskShuffler [21], Yoon et al. introduce *schedule entropy* based on Shannon Entropy [15], which does not take into account that an attacker could observe information about previous scheduling decisions, helping predict future arrivals and is therefore unsuitable to gauge how much the employed randomization impedes an attacker. As a result, Nasri et al. [11] propose the use of conditional entropy to model an attacker's knowledge gained from observing the past. In TaskShuffler++ [20], Yoon et al. react to this criticism by explicitly stating that they do not consider an attacker that knows the entire scheduler state (*strong attacker*) and improve their schedule entropy, creating *schedule min-entropy*; yet they still leave past observations out of consideration, even though a *weak(est) attacker* (i.e. with partial timing information) also has some knowledge about the past, e.g. by measuring timestamps when itself was running. In REORDER [2], Chen et al. introduce their own, different schedule entropy based on approximate entropy, which takes past information into account; analyzing the accuracy of this approach would go beyond the scope of this paper but stays an interesting detail to investigate for future work.

In a nutshell, to the best of the author's knowledge, solely relying on Yoon et al.'s schedule (min-)entropy as a security metric can lead to an overestimation of current randomization techniques' trustworthiness.

Even if shuffling works (i.e. you can achieve **near perfect randomization of the schedule** – for analytical upper bounds, see [18]), this **might not be desirable at all**, as it could increase success rates or even allow for attacks otherwise not possible: Perfect randomization implies that any conceivable execution order occurs at some point in time, all of them with relatively equal probability (= larger attack surface). Consider a lower priority attacker task whose aim is to manipulate input read by the victim. To achieve this, it needs to run immediately before the input processing victim task (having the highest priority); this is an anterior attack scenario as described in [11]. Depending on the task structure, in an FP-environment, such a constellation would never happen since the input processing task has the highest priority, it immediately consumes the data before any other task gets a chance to manipulate it. With shuffling, however, either a very well-informed attacker can perform a targeted manipulation when he knows the victim task is next, or he just keeps on trying (brute-force) and at some point in time, the randomization will do the job for him (see Figure 3).

Depending on the type of attack – anterior, posterior (e.g. modify written values after control task), pincer (e.g. prepare side-channel before and probe side-channel after victim task) – and individual system parameters, a thorough (preliminary) analysis of potential counterproductive (side) effects is essential [11].

3.4 Isolation beats Obfuscation?

With deadline-guarantees being the main reason to use an RTOS in the first place and scheduling being an integral part of any RTOS, caution when modifying the fundamental scheduling structure is certainly advisable. Why not learn from a different class of operating systems that has to deal with frequent attacks for decades: general purpose operating systems like Linux or Windows NT; two of their key principles that – in conjunction – keep us (relatively) safe on a day-to-day basis are isolation[5] and limiting permissions: They play a central role in Ward et al.'s recommendations for (security-critical) next-generation cyber-physical systems [19].

[5]Even the primary creator of TaskShuffler, Man-Ki Yoon, has recently contributed to system design research dealing with temporal isolation [9].

One option would be to reduce contact between different applications, especially with respect to their trustworthiness and importance to critical system functions. Firstly, providing **strong temporal and physical isolation of tasks becomes easier with multicore** architectures that use partitioned scheduling (i.e. each task is assigned to a fixed core, no migrations between cores are allowed): Interfering on or gaining information about an application running on a separate core with its dedicated cache is considerably harder than finding a side-channel or maliciously affecting the (cache) state of the one multiplexed CPU core both attacker and victim run on – microkernels (such as seL4 [6]) can also play a role in helping separate parts of the system from one another.

Secondly, **memory isolation** – an obvious feature of general purpose OSes – can dramatically reduce attack vectors when a task is already being controlled by the attacker (as it cannot modify another task's data to provoke unintended behavior or spy on it).

Alongside with memory isolation, one can introduce a less intrusive and more practical moving target defense mechanism: address space layout randomization (**ASLR**). Introducing randomness to the schedule as in TaskShuffler, etc. means mixing up the order of what task the CPU is working on in the current clock cycle. The number of clock cycles available per second is rather limited, however, in comparison to *virtual* memory addresses that are not required to all be physically available; in other words: ASLR's space of possible values to work with is orders of magnitude larger.

Limiting permissions in a fine-grained manner means limiting access to peripheral I/O, networking etc. so that an infected task can only access (and possibly damage) those components that it was granted permission for at design time. The attack scenario in Section 3.3 (of an attacker modifying input data intended for a critical task) would be nullified since the attacker task is not permitted read-/write access to that specific I/O device. Constraining yourself to just one, hierarchical permission structure (e.g. privileged vs. unprivileged mode) would impair the effectiveness of checking permissions; optimally, each task is individually assigned a set of resources it can access.

4 CONCLUSION

In this paper, we have seen realistic attack scenarios on real-time systems, so investment in security measures is inevitable. Schedule randomization on first sight appears as a tempting cure-all; the detailed scrutiny of suggested measures, however, uncovers unexpected deficiencies: High security requirements of mission-critical RTS are typically accompanied by high stability requirements (Section 3.2: "Sacrificing Stability for Security?"), restrictive prerequisites provoke an increasingly laborious and costly development and deployment process (Section 3.1), and might even aid and abet the attacker (Section 3.3: "Potentially Counterproductive Outcome").

On the contrary, a more complex OS (and consequently more complex runtime analyses) that inherently offers enhanced security out-of-the-box in terms of a combination of isolation-based defenses (Section 3.4: "Isolation beats Obfuscation?") seems more viable in the long run: "If you want to keep a secret you must also hide it from yourself. You must know all the while that it is there, but until it is needed you must never let it emerge into your consciousness in any shape that could be given a name" [13].

REFERENCES

[1] Hyeongboo Baek and Chang Mook Kang. 2020. Scheduling Randomization Protocol to Improve Schedule Entropy for Multiprocessor Real-Time Systems. *Symmetry* 12 (2020), 753.

[2] Chien-Ying Chen, Monowar Hasan, AmirEmad Ghassami, Sibin Mohan, and Negar Kiyavash. 2018. REORDER: Securing Dynamic-Priority Real-Time Systems Using Schedule Obfuscation. *ArXiv* abs/1806.01393 (2018).

[3] Chien-Ying Chen, Sibin Mohan, Rodolfo Pellizzoni, and Rakesh B. Bobba. 2020. On Scheduler Side-Channels in Dynamic-Priority Real-Time Systems. *ArXiv* abs/2001.06519 (2020).

[4] Chien-Ying Chen, Sibin Mohan, Rodolfo Pellizzoni, Rakesh B. Bobba, and Negar Kiyavash. 2019. A Novel Side-Channel in Real-Time Schedulers. *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2019), 90–102.

[5] Gerhard Fohler. 1995. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. *Proceedings 16th IEEE Real-Time Systems Symposium* (1995), 152–161.

[6] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *SOSP '09*.

[7] Kristin Krüger, Marcus Völp, and Gerhard Fohler. 2018. Vulnerability Analysis and Mitigation of Directed Timing Inference Based Attacks on Time-Triggered Systems. In *ECRTS*.

[8] Chang Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20 (1973), 46–61.

[9] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. 2019. Virtual timeline: a formal abstraction for verifying preemptive schedulers with temporal isolation. *Proc. ACM Program. Lang.* 4 (2019), 20:1–20:31.

[10] Sibin Mohan, Man-Ki Yoon, Rodolfo Pellizzoni, and Rakesh Bobba. 2014. Real-Time Systems Security through Scheduler Constraints. *2014 26th Euromicro Conference on Real-Time Systems* (2014), 129–140.

[11] Mitra Nasri, Thidapat Chantem, Gedare Bloom, and Ryan M. Gerdes. 2019. On the Pitfalls and Vulnerabilities of Schedule Randomization Against Schedule-Based Attacks. *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2019), 103–116.

[12] NVIDIA. 2020. NVIDIA DRIVE AGX. https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware/. Accessed: 2020-06-27.

[13] George Orwell. 1949. Nineteen Eighty-Four (1984). http://www.george-orwell.org/1984. Accessed: 2020-05-09.

[14] Rodolfo Pellizzoni, Neda Paryab, Man-Ki Yoon, Stanley Bak, Sibin Mohan, and Rakesh Bobba. 2015. A generalized model for preventing information leakage in hard real-time systems. *21st IEEE Real-Time and Embedded Technology and Applications Symposium* (2015), 271–282.

[15] Claude Elwood Shannon. 1948. A Mathematical Theory of Communication. http://people.math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf. In *The Bell System Technical Journal, Vol. 27.* Accessed: 2020-05-09.

[16] Marco Spuri and Rocquencourt. 1996. Analysis of Deadline Scheduled Real-Time Systems.

[17] Emil Talpes, Atchyuth Gorti, Gagandeep S. Sachdev, Debjit Das Sarma, Ganesh Venkataramanan, Peter Joseph Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Chris Hsiong, and Sahil Arora. 2020. Compute Solution for Tesla's Full Self-Driving Computer. *IEEE Micro* 40 (2020), 25–35.

[18] Nils Vreman, Richard Pates, Kristin Krüger, Gerhard Fohler, and Martina Maggio. 2019. Minimizing Side-Channel Attack Vulnerability via Schedule Randomization. *2019 IEEE 58th Conference on Decision and Control (CDC)* (2019), 2928–2933.

[19] Bryan C. Ward, Richard Skowyra, Samuel Jero, Nathan Burow, Hamed Okhravi, Howard Shrobe, and Roger Khazan. 2019. Security Considerations for Next-Generation Operating Systems for Cyber-Physical Systems. MIT Lincoln Laboratory.

[20] Man-Ki Yoon, Jung-Eun Kim, Richard Bradford, and Zhong Shao. 2019. TaskShuffler++: Real-Time Schedule Randomization for Reducing Worst-Case Vulnerability to Timing Inference Attacks. *ArXiv* abs/1911.07726 (2019).

[21] Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen, and Lui Sha. 2016. TaskShuffler: A Schedule Randomization Protocol for Obfuscation against Timing Inference Attacks in Real-Time Systems. *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2016), 1–12.